Programmazione I

A.A. 2002-03

Il linguaggio Java

(Lezione XI)

Operatori predefiniti, Conversioni di tipo

Prof. Giovanni Gallo
Dr. Gianluca Cincotti
Dipartimento di Matematica e Informatica
Università di Catania
e-mail : {gallo, cincotti}@dmi.unict.it

Avere i dati è bello ma…

… è farci operazioni sopra quello che serve!

OPERATORI PREDEFINITI

- aritmetici,
- relazionali e logici,
- di assegnazione,
- bit a bit (bitwise).

Anche in Java non è cambiata la regola …

- Si possono "sommare" pere con banane?
  - La Maestra ci ha detto di no!
- Alla stessa maniera è importante rispettare le "*regole dei tipi*" e non fare confusione in JAVA.
  - La richiesta del rispetto di regole di omogeneità di tipo nelle operazioni tra variabili si traduce nell'affermazione in termini tecnici che "*JAVA è un linguaggio tipato*".

Espressioni aritmetiche

- Un' ***espressione*** è una combinazione di operatori ed operandi
- Un' ***espressione aritmetica*** calcola valori numerici e usa operatori aritmetici:

somma	+
sottrazione	-
moltiplicazione	*
divisione	/
resto	%

Divisione intera e resto

- Se entrambi gli operandi dell'operatore **/** sono interi, il risultato è intero e la parte decimale è persa

14 / 3	**uguale a**	**4**
8 / 12	**uguale a**	**0**

- L'operatore resto **%** riporta il resto della divisione

14 % 3	**uguale a**	**2**
8 % 12	**uguale a**	**8**

Per tutti i tipi di numeri

Operazioni tra numeri dello stesso tipo sono possibili usando:

+ , - , * , /

Il risultato è dello stesso tipo !

Se si divide per l'intero zero si genera un segnale di errore a tempo di esecuzione (detto in Java "eccezione").

Se si divide per il float o double zero si genere un infinito o un NaN.

Operazioni tra tipi di numeri diversi ?

- Sono possibili.
- Il risultato di che tipo sarà ?
  - Sarà del tipo che non comporta rischi di perdita di informazioni (*promozione*).
  - Esempi:
    - int OP long → long;
    - int OP short → int;
    - double OP long → double.

Sulla conversione tra tipi ritorneremo più avanti.

Precedenza tra operatori

- Gli operatori possono essere combinati in espressioni complesse

```
risultato = totale + cont / max - scarto;
```

- Gli operatori hanno una ***precedenza*** ben definita implicita che determina l'ordine con cui vengono valutati
  - Moltiplicazione, divisione e resto sono valutati prima di somma, sottrazione.
- Gli operatori che hanno la stessa precedenza sono valutati (*associatività*) da sinistra a destra (tranne l'assegnazione)
- Mediante le parentesi si può alterare l'ordine di precedenza

Precedenza tra operatori (cont.)

- Ordine di valutazione dell'espressione:

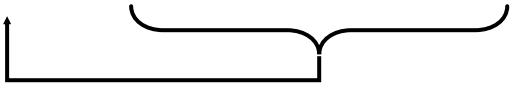
```
a + b + c + d + e          a + b * c - d / e
  1   2   3   4              3   1   4   2

a / (b + c) - d % e
  2   1   4   3

a / (b * (c + (d - e)))
  4   3   2   1
```


Operatore di assegnazione

- Ha la precedenza più bassa di qualunque altro operatore

Prima si valuta l'espressione alla destra dell'operatore =

```
risposta  =  somma / 4 + MAX * altro;
             4       1   3     2
```



Poi il risultato è assegnato alla variabile alla sinistra

Operatore di assegnazione (cont.)

- Ha la precedenza più bassa di qualunque altro operatore.
- È associativo a destra.
- È un vero e proprio operatore binario in quanto restituisce un valore !

```
x = y = z = 5;
    3   2   1
```

```
x = ( y = (z = 5) );
```


Operatori di assegnazione con operazione

- Spesso eseguiamo operazioni su una variabile, e poi memorizziamo il risultato nella stessa variabile.
  - In questo caso è possibile usare gli operatori di *assegnazione con operazione.*
    - *In pratica, si tratta di abbreviazioni.*
- Esempio:

```
totale += somma;
```

equivale a

```
totale = totale + somma;
```

Operatori di assegnazione con operazione (cont.)

operatore	esempio	equivale a
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`%=`	`x %= y`	`x = x % y`

Operatori di assegnazione con operazione (cont.)

- L'operando di destra di un operatore di assegnamento può essere un'espressione
  - L'espressione di destra viene dapprima valutata, poi il risultato è opportunamente computato con il precedente valore della variabile ed infine assegnato a quest'ultima.
- Nell'istruzione

```
risultato /= (totale-MIN) % num;
```

si calcola prima il valore dell'espressione a destra

```
((totale-MIN) % num);
```

quindi si valuta **`risultato / valore_espressione`**

e lo si assegna a **`risultato`**

Il grande favorito di tutta la storia della programmazione !

Incremento di una unità: operazione molto comune!

```
int n = 12;
n++;
```

Produce per n il valore 13.

Esiste anche una versione "prefissa":

```
int n = 12;
++n;
```

In questo esempio fa esattamente la stessa cosa.

Operatori di incremento e decremento

- Gli operatori di incremento e decremento sono operatori aritmetici unari.
  - L'operatore di *incremento* (**++**) aggiunge 1 al suo operando
  - L'operatore di *decremento* (**--**) sottrae 1 al suo operando
- L'istruzione **`somma++;`**

  equivale all'istruzione **`somma = somma + 1;`**
- Questi operatori possono essere usati in *forma prefissa* (prima della variabile) o in *forma postfissa* (dopo la variabile)

Differenza tra forma prefissa e postfissa

- Si manifesta solo se l'operatore ++ (o --) viene usato dentro altre espressioni.
  - *Prefissa*: l'incremento viene eseguito prima di usare il valore dell'operando nell'espressione;
  - *Suffissa*: l'incremento viene eseguito dopo aver usato il valore dell'operando nell'espressione.

```
int  m = 7;
int  n  = 7;
int  a  = 2 * ++m;
int  b  = 2 * n++;
```

Dopo l'esecuzione di questo codice
m=n=8,

a=16 , b=14.

Operatori relazionali

(A = = B) restituisce:
true se A e B hanno lo stesso valore,
false altrimenti

(A ! = B) restituisce:
true se A e B hanno valore differente,
false altrimenti

(A > B) restituisce:
true se A è maggiore di B
false altrimenti

(A > = B) restituisce:
true se A è maggiore o eguale a B,
false altrimenti

(A < B) restituisce:
true se A è minore di B
false altrimenti

(A < = B) restituisce:
true se A è minore o eguale di B
false altrimenti

Operatori logici

- Nelle espressioni booleane si possono usare gli *operatori logici*

!	**NOT**
&&	**AND**
||	**OR**

  - che richiedono operandi di tipo *boolean* e producono un risultato *boolean*
    - L'operatore logico NOT è un operatore unario (ha un solo operando)
    - Gli operatori logici AND e OR sono operatori binari (richiedono due operandi)

Operatore logico NOT

- L'operatore logico di negazione *NOT* è anche chiamato *complemento logico*
  - **`!a`** restituisce true se e solo se **`a`** è false.
- Il valore di un'espressione logica può essere determinato mediante la seguente *tabella di verità* :

`a`	`!a`
`true`	`false`
`false`	`true`

Gli operatori logici AND e OR

- L'espressione logica *and*

  `a && b`

  è vera se entrambi gli operandi `a` e `b` sono veri, ed è falsa altrimenti.

- L'espressione logica *or*

  `a || b`

  è falsa se entrambi gli operandi `a` e `b` sono falsi, ed è vera altrimenti.

Gli operatori logici AND e OR (cont.)

- Il valore di un'espressione logica può essere determinato mediante la seguente *tabella di verità*.
  - Poiché **`&&`** e **`||`** hanno due operandi ciascuno, ci sono 4 possibili combinazioni

`a`	`b`	`a && b`	`a || b`
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

Espressioni logiche

- L'impiego di operatori relazionali fornisce operandi booleani da combinare in espressioni logiche (condizioni) complesse.
  - `importo < totaleFattura  && !pronto`

- Gli operatori logici hanno un livello di precedenza maggiore rispetto a quelli relazionali, tuttavia per le condizioni complesse è buona regola utilizzare le parentesi
  - `(importo < totaleFattura) && (!pronto)`

Valutazione booleana corto-circuitata

(A = = B) && (B = = C)

La seconda espressione booleana viene calcolata se e solo se non si è già potuto dedurre dal calcolo della prima quale sia il risultato. Se quindi (A = = B) fosse ***false*** la successiva espressione non verrà calcolata.

(x!=0) && (1/x>1/2)

La divisione 1/x verrà calcolata solo se x risultasse non nulla.

Operatori bit a bit (bitwise)

Supponiamo di avere due int, a e b (si può applicare a tutti i tipi interi).

Internamente a e b sono rappresentati in binario:

a = 16 = 00000000 00000000 00000000 00010000

b = 24 = 00000000 00000000 00000000 00011000

a & b
restituisce l'AND bit a bit dei due numeri (ancora un numero):
00000000 00000000 00000000 00010000

a | b
restituisce l'OR bit a bit dei due numeri (ancora un numero):
00000000 00000000 00000000 00011000

a ^ b
restituisce lo XOR (OR esclusivo) bit a bit dei due numeri (ancora un numero):
00000000 00000000 00000000 00001000

Operatori bit a bit (cont.)

~a
restituisce il NOT bit a bit di a :
11111111 11111111 1111111 11101111

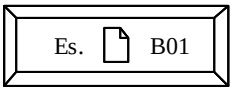
a << 4
restituisce la stringa di binari "shiftata di 4 posti verso sinistra:
00000000 00000000 00000001 00000000

a >>2
restituisce la stringa di binari "shiftata di 2 posti verso destra:
00000000 00000000 00000000 00000100

Un comune utilizzo degli operatori bit a bit : il masking

Vogliamo sapere se il terzo bit di un intero è 1 oppure 0.

```
int a = 345;
int b = 4 ;       // infatti 4 in binario è   00 … 00 00000100
int terzoBit = (a & b)/4;
```



Conversione di tipi

Se gli operatori hanno operandi non omogenei cosa succede?

Avvengono conversioni *automatiche*, tutte in modo che non si perda informazione.

Ad esempio, considerare un intero come un numero in virgola mobile:

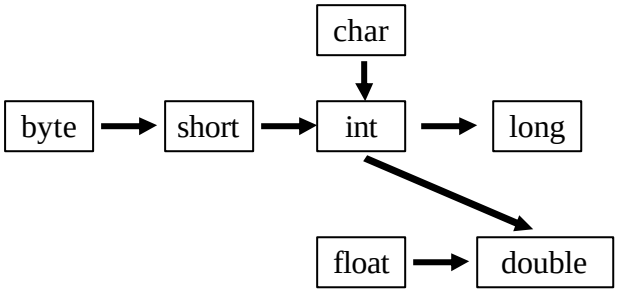
```
float  x = 3.3;
float  y = x + 2;
```

Modalità di conversione

- Le conversioni tra tipi di dato possono avvenire in tre modi:
  1) ***Promozione*** in un'espressione aritmetica;
  2) ***Conversione*** durante assegnazione;
  3) ***Casting*** esplicito.

Promozione

- Promozione automatica di tipo in un'espressione.
  - In alcune situazioni, gli operandi di operatori numerici vengono convertiti automaticamente in un tipo "superiore" sufficientemente "capiente" prima dell'azione dell'operatore, in modo da garantire la corretta esecuzione dell'operazione.



Conversione durante assegnazione

- Conversione di tipo durante un operazione di assegnazione.
  - Oltre alla promozione automatica, esistono altre situazioni in cui un valore di un certo tipo viene convertito in un tipo diverso, per esempio durante un assegnazione o un'inizializzazione.
  - Esempio:
    - byte b = 42; // conversione implicita da int a byte (narrowing)
    - float x = b; // conversione implicita da byte a float (widening)

Casting

- Per forzare esplicitamente una conversione di tipo, si usa l'operatore di cast a tipo, (*tipo*), seguito dall'espressione il cui valore deve essere convertito al tipo *tipo:*

  (*tipo*) espressione;

- Per definizione il cast inibisce il controllo sui tipi operato dal compilatore!
  - Si possono realizzare sia conversioni *widening* che *narrowing*.

Casting (cont.)

- Esempio:

```
int  i,  j;
float  x = 123.4e10f;
i = x;              // errore: int e float sono tipi diversi
j = (int) x;      // nessun avvertimento !
```

- Esempio:

```
int x, y;
float quoziente = x / y;      // Errore logico
float quoziente = (float) x / y;
// oppure  quoziente = x / (float) y;
```


Fine